Journal of Multidisciplinary Engineering
Published by DigitoPub | Digitodontics International Academy
DigitoPub
Open Access & Peer-reviewed
Indexed in: Crossref, ICI, Google Scholar

Why TCP Reno Collapses over Noisy Wireless Links: A Simulation, Analytical-Model and Emulation-Based Evaluation

Tahani E. Alqunsul[1] | Fatema M. Shalouf[1]

[1]Department of Electrical and Computer Engineering, Engineering College, Nalut University, Jado, Libya.

ABSTRACT

Purpose: To quantify how increasing channel noise, expressed as packet drop probability and the equivalent bit error rate (BER), degrades TCP Reno during a bulk FTP transfer, and to identify the mechanisms behind the degradation.

Materials and Methods: A client–IP cloud–server topology was simulated in OPNET Modeler 14.5, and a 10 MB file was downloaded under five drop probabilities from 0% to 10% (BER up to about 10^{-5}). Results were compared with a published reference study, with closed-form expectations for datagram count, retransmissions and throughput bounds, with the Mathis and Padhye steady-state models, and with a virtualised Linux testbed.

Results: Response time stayed at 10.08 s up to 0.1% drop, rose 2.6-fold at 1% (26.01 s) and 191-fold at 10% (1,926.5 s), while effective goodput fell from about 992 kB/s to 5.2 kB/s. At 10%, 1,015 retransmissions were observed against 761 expected from independent losses. The Mathis model matched the 1% case within 18% but overestimated the 10% goodput nineteen-fold.

Conclusions: Degradation is strongly nonlinear and is dominated by window collapse and retransmission timeouts rather than by the raw number of lost packets. Default TCP Reno should not be used on links with drop probabilities of about 1% or more without link-layer recovery or loss-tolerant alternatives. *(J Multidiscip Eng 2026;1:94-101)*

Keywords: TCP Reno, bit error rate, packet drop probability, congestion control, OPNET simulation, wireless networks, FTP performance.

ARTICLE INFO

*Corresponding author:
Tahani E.Alqunsul
Email: t.alqunsul@nu.edu.ly

Received: 22-08-2026
Accepted: 27-10-2026
Published: 29-10-2026

DOI:
https://doi.org/10.26629/jme.2026.12

Copyright: © 2026 by the authors.
Articles published in the *J Multidiscip Eng* are licensed under a Creative Commons Attribution-Noncommercial 4.0 International License.
https://creativecommons.org/licenses/by-nc/4.0/

Audioabstract:

How to cite this article | Alqunsul TE, Shalouf FM Why TCP Reno collapses over noisy wireless links: a simulation, analytical-model and emulation-based evaluation. J Multidiscip Eng. 2026;1:94-101.

1. INTRODUCTION

The Transmission Control Protocol (TCP) carries most of the data exchanged over the Internet, and its flow-control and congestion-control mechanisms were designed around one central assumption: a lost packet signals buffer overflow somewhere along the path.[1,2] On wired paths, this assumption is usually reasonable. On wireless, satellite and mobile links, however, a considerable share of losses is caused by corrupted frames rather than by congestion. The sender cannot distinguish the two causes, so it still reduces its congestion window and, when the loss is severe, waits for a retransmission timeout before it resumes sending.[3,4] The resulting throughput penalty grows much faster than the loss rate itself, which makes the behaviour of TCP on noisy channels a long-standing concern for network designers.

Many remedies have been proposed, including selective acknowledgements,[5,6] delay-based congestion estimation as in TCP Vegas,[7] link-layer

retransmission and split-connection schemes,[4] newer congestion-control algorithms such as CUBIC and BBR,[8,9] and UDP-based transports.[10,11] Nevertheless, TCP Reno and its close relatives remain widely deployed, and many embedded and legacy systems still run them with default settings. Practitioners therefore need quantitative guidance on the noise level at which a default TCP configuration stops being usable and on the mechanisms that make it fail.

Sarkar et al. recently quantified this effect for a simple FTP scenario using Riverbed Modeler and a virtualised Linux testbed, and reported that throughput collapses at a 10% drop probability, with a download time about 32 times longer than on a perfect channel.[12] Reproducing such a study independently is valuable for two reasons. It tests whether the conclusions survive a change of simulator release and configuration, and it makes it possible to separate effects caused by loss from effects caused by the experimental set-up.

This study therefore re-implements the scenario in OPNET Modeler 14.5 and examines how packet drop probability, and the equivalent BER, affect FTP response time, retransmission count, congestion-window behaviour and goodput for TCP Reno with default parameters. The measurements are validated against closed-form expectations and against the steady-state models of Mathis et al. and Padhye et al.,[13,14] and are compared with the reference results. The main contributions are a side-by-side quantification of the degradation under two independent configurations, evidence that the loss-dominated regime converges to the same goodput of about 5 kB/s regardless of the baseline, and a model-based explanation of why retransmissions and delay grow nonlinearly as the channel deteriorates.

2. MATERIALS AND METHODS

This is a simulation-based experimental study. No human participants, patient data or surveys were involved, so ethical approval was not required.

2.1 Network model

The experiment was run in OPNET Modeler 14.5, whereas the reference study used Riverbed Modeler 16.0.[12] The topology was kept deliberately simple to limit the influence of unrelated parameters (Fig. 1). An FTP client (ethernet_wkstn_adv) and an FTP server (ethernet_server_adv) were each attached by a 10 Mbit/s Ethernet link to an IP cloud (ethernet4_slip8_cloud_adv) that emulates the wide-area network. The cloud packet latency was set to 0.02 s, and its packet discard ratio was varied from one scenario to the next to represent channel noise.

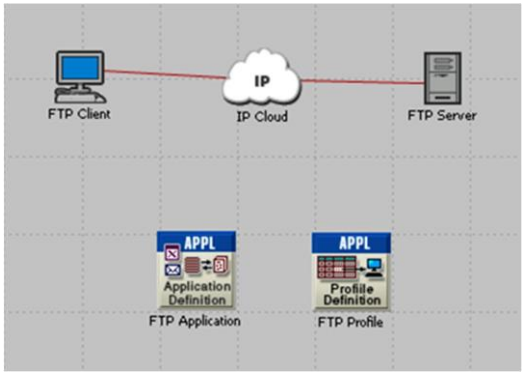


Fig. 1: Simulation topology: FTP client and FTP server connected through an IP cloud, with the application and profile definitions.

2.2 TCP and application settings

Both end nodes used the default Reno configuration: fast retransmit and Reno fast recovery enabled, duplicate-ACK threshold of 3, initial slow-start window of 2 MSS, receiver buffer of 8,760 bytes, and the maximum segment size assigned automatically (1,460 bytes). Delayed acknowledgments were limited to two segments or 0.2 s. The retransmission timer used an initial value of 3 s, a minimum of 1 s, a maximum of 64 s and a timer granularity of 0.5 s, with Karn's algorithm enabled. Selective acknowledgment, window scaling, timestamps and ECN were disabled. The FTP application issued a single download (100% get commands) of a constant 10,000,000-byte file with best-effort service, and the profile was executed once.

2.3 Scenarios and measured statistics

Five scenarios were simulated according to the packet drop probability p_{drop} (Table 1). The equivalent BER follows from the packet error rate and the packet length (Eq. 1), which is why a drop probability of 10% corresponds to a BER of about

10^{-5} for packets of roughly 1,250 bytes. Each run lasted 2,700 s, and 2,700 samples were collected for seven statistics: FTP download response time, server retransmission count, server traffic load, server congestion window, client sent-segment ACK number, client throughput and client receiver window.

Table 1: Simulated scenarios, with the equivalent BER and channel condition.

Scenario	p_{drop} (%)	BER	Channel condition
S1	0	0	Perfect channel
S2	0.01	≈10^{-8}	Low-noise channel
S3	0.1	≈10^{-7}	Medium-noise channel
S4	1	≈10^{-6}	High-noise channel
S5	10	≈10^{-5}	Very-high-noise channel

2.4 Validation and statistical treatment

Four validation steps were applied. First, the number of datagrams and retransmissions was compared with closed-form expectations. Second, the achieved throughput was compared with the window-limited and medium-limited bounds. Third, the download time was recomputed from those bounds. Fourth, the behaviour was cross-checked in a virtualised testbed of two Linux hosts connected through a Linux router that dropped packets at a configured probability. Effective goodput was derived as the file size divided by the response time. Each scenario was executed once with the default random seed, so the results are single-run observations; no confidence intervals or P values are reported, and differences are expressed as ratios and relative errors.

3. THEORY AND CALCULATION

3.1 Loss model

For independent bit errors, a packet of L bits is corrupted when at least one bit is in error, so the packet error rate (PER) is

$$\mathrm{PER} = 1 - (1 - \mathrm{BER})^{L} \approx L \cdot \mathrm{BER} \qquad (1)$$

which reduces to PER ≈ L·BER for small BER. A 1,250-byte packet (10,000 bits) thus has a PER of 10% at a BER of 10^{-5}.

3.2 Reno dynamics

TCP Reno starts in slow start and doubles its window every round-trip time (RTT) until the slow-start threshold is reached, after which congestion avoidance increases the window by about one MSS per RTT.[3] Three duplicate acknowledgments trigger fast retransmit; the threshold and window are then roughly halved. Reno fast recovery repairs only one lost segment per window, so several losses within one window usually end in a retransmission timeout, after which the window collapses to one segment and the timer doubles at every further failure.[15,16] Recovering from a single halving of a window of W segments takes W/2 round trips, and a timeout costs at least the minimum RTO of 1 s, which is 25 times the 0.04 s RTT of the present topology.

3.3 Expected datagrams and retransmissions

Transferring S = 10,000,000 bytes with an MSS of 1,460 bytes requires n = S/MSS = 6,849.3 segments. If every transmission is lost independently with probability p, the number of transmissions per segment is geometric with mean 1/(1−p), so the expected number of retransmissions is

$$N_{\mathrm{rtx}} = \frac{n \cdot p}{1 - p} \qquad (2)$$

which exceeds the lower bound n·p because retransmitted segments can themselves be lost.

3.4 Throughput bounds

Throughput is limited by the receiver window and by the medium. With a receiver window rwnd, the window-limited throughput is

$$Th_{w} = \frac{\mathrm{rwnd}}{\mathrm{RTT}} \qquad (3)$$

For a link of capacity C, each segment carries H = 78 bytes of overhead (20 TCP, 20 IP, 18 Ethernet, and 20 preamble and inter-frame gap), so the medium-limited throughput is

$$Th_{\mathrm{eth}} = \frac{C}{8} \cdot \frac{\mathrm{MSS}}{\mathrm{MSS} + H} \qquad (4)$$

The achieved throughput is approximately the

smaller of the two, and the transfer time is S divided by that value.

3.5 Steady-state loss–throughput models

Under periodic losses handled entirely by fast retransmit, Mathis et al. obtained[13]

$$B = \frac{\text{MSS}}{\text{RTT}}\sqrt{\frac{3}{2bp}} \quad (5)$$

where b is the number of segments acknowledged by each ACK (b = 2 with delayed acknowledgments). Padhye et al. extended the model to include retransmission timeouts[14] and obtained

$$B(p) = \frac{\text{MSS}}{D_c + D_t} \quad (6)$$

$$D_c = \text{RTT} \cdot \sqrt{\frac{2bp}{3}} \quad (7)$$

$$D_t = T_0 \min\left(1, 3\sqrt{\frac{3bp}{8}}\right) p(1 + 32p^2) \quad (8)$$

where T_0 is the mean duration of the first timeout. The first model ignores timeouts, whereas the second one includes them and is therefore expected to hold at high loss rates.

4. RESULTS AND DISCUSSION

4.1 FTP response time

Table 2 and Fig. 2 show the time needed to download the 10 MB file. In both studies the response time increases monotonically with p_{drop}. In this study it remained at 10.08 s for 0% to 0.1%, which we interpret as no measurable change at the resolution of the statistic, rose 2.6-fold at 1%, and reached 1,926.5 s at 10%, a 191-fold slowdown. The reference study shows the same pattern, with a 32-fold slowdown at 10%.[12] The two absolute baselines differ six-fold (10.08 s against 60.21 s), but the 10% results differ by only 1.4% (1,926.5 s against 1,953.61 s). When losses dominate, the completion time is therefore governed by the loss process and not by the link or the window, so differences in set-up are largely washed out.

Table 2: FTP download response time for a 10 MB file and slowdown relative to the lossless case.

p_{drop} (%)	This study (s)	Slow-down (×)	Ref. (s)	Slow-down (×)
0	10.08	1.00	60.21	1.00
0.01	10.08	1.00	60.42	1.00
0.1	10.08	1.00	61.46	1.02
1	26.01	2.58	76.83	1.28
10	1,926.5	191.0	1,953.61	32.4

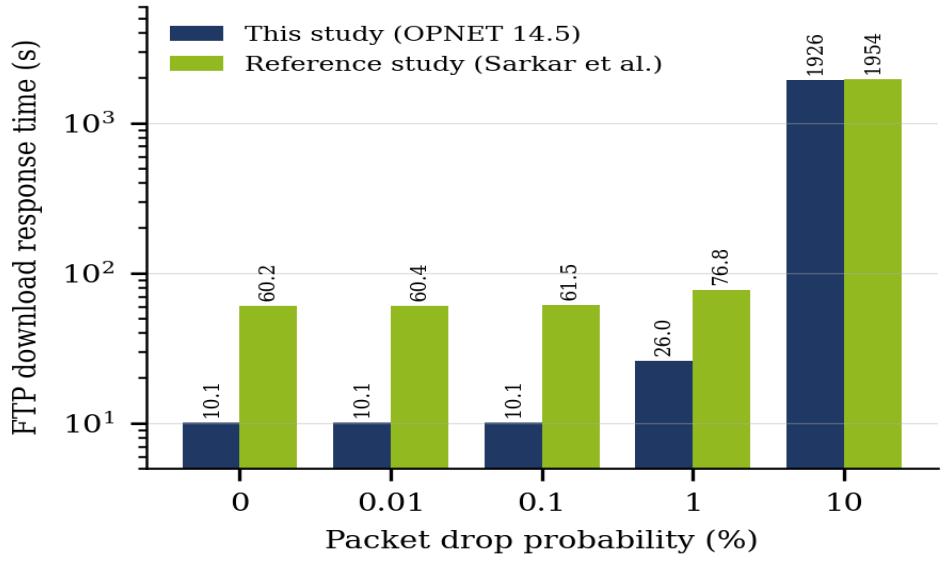


Fig. 2: FTP download response time (logarithmic scale) for the five drop probabilities in this study and in the reference study.

4.2 Retransmissions

Retransmissions grow with p_{drop} (Table 3). At 0.1% and 1% the observed counts (5 and 65) are close to the expectation $n \cdot p/(1-p)$ of 6.9 and 69.2 and lie within the sampling noise of a Poisson count (about half a standard deviation at 1%). At 10% the observed 1,015 retransmissions exceed the expected 761 by 33% (the reference study recorded 1,092, or 43% more), a deviation of roughly nine standard deviations under the same approximation, so it is systematic rather than random. Reno without selective acknowledgment explains this excess. Segments that the receiver already holds are resent after a timeout, only one hole is repaired per round trip during fast recovery, and some retransmissions are spurious.[5,6] The number of retransmissions is therefore not proportional to the drop probability, and the proportionality breaks down exactly in the regime where the delay explodes.

Table 3: Retransmission count at the FTP server compared with the expectation $n \cdot p/(1-p)$, n = 6,849.

p_{drop} (%)	Expected	This study	Ref.	Observed / expected
0	0	0	0	–
0.01	0.7	0	1	0.00
0.1	6.9	5	7	0.73
1	69.2	65	62	0.94
10	761.0	1,015	1,092	1.33

4.3 Congestion window and acknowledgment progress

Fig. 3 shows the server congestion window and the client ACK number in the 10% scenario. The window never exceeds about 11,700 bytes (8 MSS), and it repeatedly collapses to about 1,460 bytes (1 MSS), the signature of retransmission timeouts rather than of the moderate halvings typical of fast recovery. With a mean window of about 2 MSS and an RTT of 0.04 s, the window alone would allow roughly 73 kB/s, whereas the measured goodput is 5.2 kB/s. By this rough estimate the connection transmits during only about 7% of the elapsed time and idles for the rest while it waits for timers to expire. The ACK number climbs slowly and almost steadily by about 10 million bytes and flattens at roughly 1,900–2,000 s, when the transfer ends and the window freezes at a constant residual value. Because the window cannot grow between successive losses, the sender rarely holds enough segments in flight to generate three duplicate acknowledgments, and each loss is therefore more likely to end in a timeout.

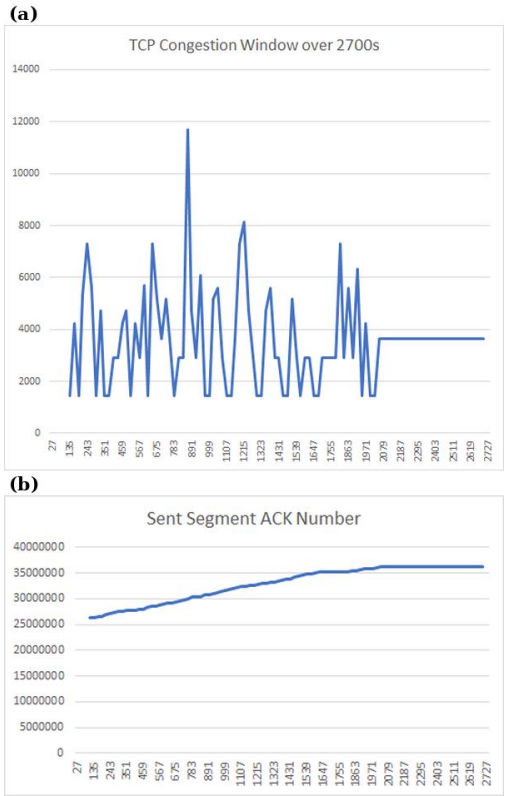


Fig. 3: Effect of packet loss at p_{drop} = 10%: (a) TCP congestion window (bytes) and (b) sent-segment ACK number of the client over 2,700 s of simulation time.

4.4 Goodput and comparison with analytical models

Figure 4 and Table 4 compare the measured

goodput with the two steady-state models, evaluated for RTT = 0.04 s (twice the cloud latency), MSS = 1,460 bytes and b = 2 and capped at the Ethernet limit. T_0 is treated as a plausible range of 1–3 s because the minimum RTO is 1 s, the initial RTO is 3 s, and the timer doubles after repeated failures. The two models describe different regimes. At 1% the Mathis model underestimates the measured goodput by 18%, whereas the Padhye model with timeouts underestimates it by 47% to 69%, which shows that most losses were still repaired by fast retransmit. At 10% the situation reverses. The Mathis model overestimates the goodput nineteen-fold (100.0 against 5.19 kB/s), whereas the Padhye model with T_0 = 3 s gives 4.29 kB/s, within 17% of the measurement. The transition between 1% and 10% is therefore the point at which timeouts take over as the dominant cause of delay.

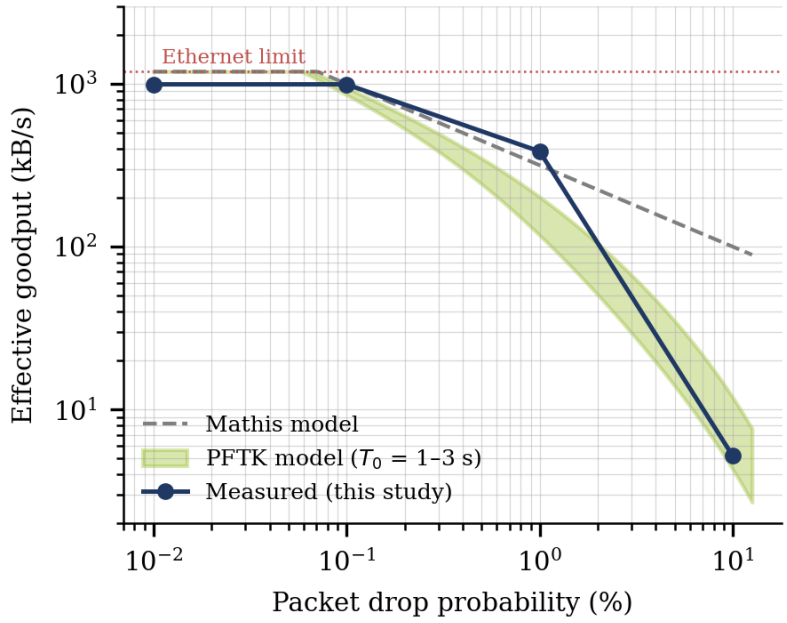


Fig. 4: Measured goodput compared with the Mathis and Padhye (PFTK) models, capped at the Ethernet limit.

Table 4: Measured and modelled goodput (kB/s). Models are capped at the Ethernet limit of 1,186.6 kB/s.

p_{drop} (%)	Measured	Mathis	PFTK, T_0 = 1 s	PFTK, T_0 = 3 s
0.01	991.6	1,186.6	1,186.6	1,186.6
0.1	991.6	999.6	946.4	855.3
1	384.4	316.1	202.1	117.4
10	5.19	100.0	11.86	4.29

4.5 Analytical cross-checks

Table 5 summarises the closed-form checks. The 6,854 frames sent in the lossless case differ from the 6,849.3 calculated segments by only 0.07%, and the small excess corresponds to the TCP handshake, the connection teardown and address resolution. The reference baseline of 200 kB/s agrees with the window-limited bound of 219 kB/s (rwnd/RTT = 8,760 bytes / 0.04 s) within 9%, which identifies the reference configuration as window-limited, with a calculated time of 45.66 s against the observed 60.21 s. The baseline of this study (991.6 kB/s) is instead 84% of the Ethernet limit, which identifies it as medium-limited, with 10.08 s observed against 8.43 s calculated. Both gaps (32% and 20%) are plausibly caused by slow

start, delayed acknowledgments and connection set-up, which the simple bounds ignore. The 8,760-byte receiver buffer was configured identically in both studies, so the reason why the window did not constrain this study's baseline is not resolved here, and differences between the simulator releases remain a plausible but unverified explanation.[12]

Table 5: Closed-form checks against observed values.

Quantity	Calculated	Observed	Difference
Segments for 10 MB	6,849.3	6,854	+0.07%
Window-limited throughput (kB/s)	219.0	200.0 (ref.)	−8.7%
Time, window-limited (s)	45.66	60.21 (ref.)	+31.9%
Ethernet-limited throughput (kB/s)	1,186.6	991.6	−16.4%
Time, Ethernet-limited (s)	8.43	10.08	+19.7%

In the virtualised testbed the download time showed the same behaviour as in simulation. Low drop probabilities had little influence, a probability of 1% lengthened the download clearly, and 10% produced a very long download. A packet capture at 1% showed that the connection needed about 27 round trips to recover from one drop, and the resulting overhead was calculated as 29.0 s. This is the same order of magnitude as the 15.9 s increase simulated at 1% in this study and the 16.6 s increase in the reference study, which is consistent with the delay at 1% being caused mainly by the slow regrowth of the window and not by the retransmitted bytes themselves.

4.6 Limitations and implications

The results rest on single runs, a single traffic type, and uniformly distributed drops. Real wireless errors are bursty, and bursts would change the balance between fast retransmit and timeouts. The default settings used here also disable selective acknowledgement and window scaling, so the figures describe a worst-case Reno rather than a modern stack. Within these limits, the evidence supports three practical conclusions. Link-layer error control should keep the residual drop probability seen by TCP well below 1%. Selective acknowledgment, a lower minimum RTO or loss-tolerant congestion control should be tested on links that cannot meet that target.[9] Where reliability is only needed at the application layer, UDP-based or datagram-oriented transports are alternatives.[10,11] These measures were not evaluated here and remain subjects for future work, including IoT and emergency-communication scenarios.

5. CONCLUSION

This study re-implemented an FTP download over a noisy channel in OPNET Modeler 14.5 and analysed TCP Reno with simulation, closed-form calculation, steady-state models and a virtualised testbed. The download time was unaffected up to a drop probability of 0.1%, increased 2.6-fold at 1% and 191-fold at 10%, where goodput fell to about 5 kB/s, the same value that the reference study reached from a six-fold slower baseline. Retransmissions followed the independent-loss expectation up to 1% but exceeded it by a third at 10%, and the congestion window repeatedly collapsed to one segment, which indicates that retransmission timeouts rather than the number of lost packets cause the collapse. The Mathis model described the 1% case and the Padhye model with timeouts described the 10% case, which locates the transition between the two regimes between these values. The findings are limited by single simulation runs, uniform packet drops and default TCP settings with selective acknowledgment disabled. Within those limits, TCP Reno should not be operated unprotected on links with drop probabilities of about 1% or more, and future work should evaluate selective acknowledgment, loss-tolerant congestion control and alternative transports under bursty error models, with several repetitions per scenario to provide confidence intervals.

ACKNOWLEDGEMENT

The authors would like to thank Dr. El Bhlul Fgee for his supervision and guidance during the Quality of Service module at the Libyan Academy (Zintan Branch), and the developers of the OPNET Modeler simulation tool used in this work.

AUTHOR CONTRIBUTIONS

All authors contributed equally for: Conceptualization, methodology, writing – original draft, investigation, data curation, validation, manuscript preparation, critical review, writing – review & editing, and final approval of the manuscript.

CONFLICT OF INTEREST

The authors declare no conflicts of interest concerning the research, authorship, and publication of this article.

FUNDING

This research received no external funding.

REFERENCES

1. Postel J. Transmission Control Protocol. RFC 793. Internet Engineering Task Force; 1981.
2. Jacobson V. Congestion avoidance and control. ACM SIGCOMM Comput Commun Rev. 1988;18(4):314-29.
3. Allman M, Paxson V, Blanton E. TCP congestion control. RFC 5681. Internet Engineering Task Force; 2009.
4. Balakrishnan H, Padmanabhan VN, Seshan S, Katz RH. A comparison of mechanisms for improving TCP performance over wireless links. IEEE/ACM Trans Netw. 1997;5(6):756-69.
5. Fall K, Floyd S. Simulation-based comparisons of Tahoe, Reno and SACK TCP. ACM SIGCOMM Comput Commun Rev. 1996;26(3):5-21.
6. Mathis M, Mahdavi J, Floyd S, Romanow A. TCP selective acknowledgment options. RFC 2018. Internet Engineering Task Force; 1996.
7. Brakmo LS, Peterson LL. TCP Vegas: end to end congestion avoidance on a global Internet. IEEE J Sel Areas Commun. 1995;13(8):1465-80.
8. Ha S, Rhee I, Xu L. CUBIC: a new TCP-friendly high-speed TCP variant. ACM SIGOPS Oper Syst Rev. 2008;42(5):64-74.
9. Cardwell N, Cheng Y, Gunn CS, Yeganeh SH, Jacobson V. BBR: congestion-based congestion control. Commun ACM. 2017;60(2):58-66.
10. Iyengar J, Thomson M, editors. QUIC: a UDP-based multiplexed and secure transport. RFC 9000. Internet Engineering Task Force; 2021.
11. Kohler E, Handley M, Floyd S. Datagram Congestion Control Protocol (DCCP). RFC 4340. Internet Engineering Task Force; 2006.
12. Sarkar NI, Ammann R, Zabir SMS. Analyzing TCP performance in high bit error rate using simulation and modeling. Electronics. 2022;11(14):2254. doi:10.3390/electronics11142254
13. Mathis M, Semke J, Mahdavi J, Ott T. The macroscopic behavior of the TCP congestion avoidance algorithm. ACM SIGCOMM Comput Commun Rev. 1997;27(3):67-82.
14. Padhye J, Firoiu V, Towsley D, Kurose J. Modeling TCP throughput: a simple model and its empirical validation. ACM SIGCOMM Comput Commun Rev. 1998;28(4):303-14.
15. Paxson V, Allman M, Chu J, Sargent M. Computing TCP's retransmission timer. RFC 6298. Internet Engineering Task Force; 2011.
16. Henderson T, Floyd S, Gurtov A, Nishida Y. The NewReno modification to TCP's fast recovery algorithm. RFC 6582. Internet Engineering Task Force; 2012.